Projet

1 Consignes générales

Ce projet devra être réalisé en binôme avec un membre de votre groupe de TP. Contactez votre chargé de TP si vous n'avez pas de binôme.

Une archive `projet.zip` est disponible sur la page Madoc du cours. Vous devrez renommer le répertoire de cette archive en **Projet_Nom1_Nom2**, avec **Nom1** et **Nom2** les noms de famille des deux membres du binôme, par ordre alphabétique. L'archive complétée avec le code OCaml du projet devra être rendu au plus tard le **dimanche 23 mars à 23 :59** sur Madoc. Chaque fichier de l'archive devra contenir un entête en commentaire avec les noms, prénoms et numéro de groupe des deux membres du projet. Vous devrez notamment fournir des nouveaux jeux de tests pour vérifier que votre vérificateur de type et votre évaluateur fonctionnent correctement. De plus, il est attendu que votre code soit commenté. Vous pourrez notamment expliquer dans les commentaires de votre code les choix de design faits. **Les membres du binôme doivent être les auteurs de l'intégralité du code de leur projet.**

Un **premier jalon**, détaillé dans la Section 6, devra être présenté à votre chargé de TP lors de votre séance de TP de **la semaine du 10 mars**. Ce premier jalon comptera pour 2 points dans la note du projet.

Une **évaluation orale** du projet produit par chacun des binômes aura lieu lors de votre séance de TP de **la semaine du 24 mars**. Lors de cette courte évaluation, votre chargé de TP vous interrogera sur le code de votre projet, et sur les choix de design faits.

2 Sujet

Le projet porte sur l'implémentation d'un vérificateur de type et d'un évaluateur pour un langage de programmation appelé `SimpleML` . L'architecture du projet est la suivante :

- Les analyseurs lexical et syntaxique de ce langage vous sont fournis, respectivement dans les fichiers `lexer.mll` et `parser.mly`.
- Les définitions des arbres de syntaxe abstraite des différentes constructions de votre langage vous sont fournies dans le fichier `syntax.ml`.
- Le code OCaml de la vérification de type sera à compléter dans le fichier `verif.ml`.
- Celui de l'évaluateur sera à compléter dans le fichier `evaluateur.ml`.
- Un fichier dune vous est fourni. Vous pouvez donc compiler votre projet en utilisant `dune build`.
- Le binaire généré par le projet est définie dans le fichier `eval.ml`. Il s'utilise avec la commande `dune exec -- ./eval.exe nomdufichier` où `nomdufichier` est un fichier avec le programme `SimpleML` à évaluer.

La réalisation de la vérification de type et de l'évaluateur pour le fragment de base de `SimpleML` sera évaluée sur 16. Vous pourrez ensuite choisir de réaliser certaines extensions du langage détaillées dans la Section 7 pour obtenir les 4 points supplémentaires.

3 Syntaxe

Il y a quatre constructions principales dans la syntaxe de ce langage : les types simples, les expressions, les déclarations de fonctions, et les programmes.

Les types de `SimpleML` sont soit `int` pour les entiers, soit `bool` pour les booléens. Le type OCaml des types de `SimpleML` s'appelle `typ`, il est fourni dans le fichier `syntax.ml`. Dans la suite, on note σ, τ pour représenter un type de `SimpleML` .

Les expressions sont définies par la syntaxe suivante :

$$\begin{aligned}\mathtt{M},\mathtt{N} \triangleq\ & x \mid n \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{M}+\mathtt{N} \mid \mathtt{M}*\mathtt{N} \mid \mathtt{M}-\mathtt{N} \mid \mathtt{M}/\mathtt{N} \mid \mathtt{M}=\mathtt{N} \mid \mathtt{M}<\mathtt{N} \mid \mathtt{if}\ \mathtt{M}\ \mathtt{then}\ \mathtt{N}_1\ \mathtt{else}\ \mathtt{N}_2 \mid \\ & \mathtt{not}\ \mathtt{M} \mid \mathtt{M}\ \mathtt{and}\ \mathtt{N} \mid \mathtt{M}\ \mathtt{or}\ \mathtt{N} \mid \mathtt{f}(\mathtt{M}_1,\ldots,\mathtt{M}_k) \mid \mathsf{let}\,(x:\sigma)=\mathtt{M}\,\mathsf{in}\,\mathtt{N}\end{aligned}$$

avec x un identifiant de variable, `f` un identifiant de fonction, et $n \in \mathbb{Z}$ un entier.

La syntaxe des expressions est donc formée par :

— les opérateurs arithmétiques et booléens, y-compris if-then-else.

— les appels de fonctions de la forme $\mathtt{f}(\mathtt{M_1},\ldots,\mathtt{M_n})$

— de déclarations locales de variables sous la forme de $\mathsf{let}\,(x:\sigma)=\mathtt{M}\,\mathsf{in}\,\mathtt{N}$ qui définit dans `N` une variable x de type σ valant la valeur résultant de `M`.

Le type OCaml des arbres de syntaxe abstraits des expressions s'appelle `expr`, il est fourni dans le fichier `syntax.ml`.

Les déclarations de fonctions sont de la forme

$$\mathtt{let}\ \mathtt{f}(x_1:\sigma_1,\ldots,x_n:\sigma_n):\tau=\mathtt{M}$$

Cela permet de déclarer une fonction prenant n arguments $x_1,\ldots,x_n$ de type respectivement $\sigma_1,\ldots,\sigma_n$, et dont le corps est `M`, qui doit produire une valeur de type τ.

Le type OCaml des arbres de syntaxe abstraits des expressions s'appelle `decl_fun`, il est fourni dans le fichier `syntax.ml`.

Un programme est formé d'une séquence de déclarations de fonctions, dont une fonction spécifique, s'appelant main, qui sera de la forme

$$\mathtt{let}\ \mathit{main}():\sigma=\mathtt{M}$$

Le type OCaml des arbres de syntaxe abstraits des programmes s'appelle `prog`, il est fourni dans le fichier `syntax.ml`.

4 Vérification de types

Dans cette première phase, l'objectif est de vérifier le type des expressions et des fonctions, pour s'assurer qu'elles soient bien typées. On vérifiera ainsi que :

— chaque variable utilisée est bien définie auparavant, et que son type est respecté ;

— que chaque appel de fonction est fourni avec des arguments du bon type, et avec le bon nombre d'arguments ;

— qu'une fonction main est bien déclarée, ne prenant aucun argument.

Cette vérification de type sera implémentée par une fonction OCaml `verif_prog` prenant en argument un programme `SimpleML` et renvoyant soit vrai si ce programme est bien typé, soit faux si ce n'est pas le cas.

Pour cela, vous implémenterez deux fonctions :

— une fonction `verif_expr` vérifiant qu'une expression a un type donné ;
— une fonction `verif_decl_fun` vérifiant qu'une déclaration de fonction est bien typée.

Dans les deux cas, vous aurez besoin de leur fournir un environnement de typage fournissant le type des variables et des identifiants de fonctions pouvant apparaître dans la portée d'une expression ou d'une déclaration de fonctions.

5 Évaluateur de code

Dans cette deuxième phase, l'objectif sera d'évaluer le code d'un programme `SimpleML` bien typé. Pour cela, vous évaluerez la fonction $main()$, qui pourra elle-même appeler d'autres fonctions de votre programme `SimpleML` .

Cette évaluation de programme sera implémentée par une fonction OCaml `eval_prog` prenant en paramètre un programme et affichant la valeur produite par l'évaluation de la fonction $main()$ de ce programme, qui sera soit un entier, soit un booléen.

Pour cela, vous implémenterez une fonction `eval_expr` prenant en paramètre une expression et fournissant la valeur résultant de son évaluation.

6 Premier jalon

Lors de la séance de TP de la semaine du 17 mars, vous présenterez à votre chargé de TP :
— le type `env_type` des environnements de typage utilisés pour la vérification de type ;
— la signature des fonctions `verif_expr`, `verif_decl_fun` et `verif_prog` ;
— le type `env_val` des environnements utilisés pour l'évaluation des expressions ;
— la signature des fonctions `eval_expr` et `eval_prog`.

Ce premier jalon comptera pour 2 points dans la note du projet.

7 Extensions

Chacune de ces extensions nécessitera de modifier la définition de l'analyseur lexical et syntaxique dans les fichiers `lexer.mll` et `parser.mly`, et d'enrichir le type OCaml de la syntaxe `SimpleML` dans le fichier `syntax.ml`.

7.0 Unit (0 points)

Cette extension ne rapporte pas de point seule, mais est requise pour plusieurs autres extensions. Vous rajouterez un nouveau type `unit` au type `typ` des types de `SimpleML` ; ainsi que l'opérateur de séquencement `M; N` permettant d'évaluer `M` avant `N`.

7.1 Flottants (2 points)

Vous rajouterez un nouveau type `float` au type `typ` des types de `SimpleML` , de même que des flottants au type des expressions `expr`. Vous rajouterez également les opérations arithmétique standard sur les flottants. Vous pourrez :
— soit rajouter des opérateurs spécifiques $+.$, $-.$, $*.$, $/.$ comme en OCaml,

— soit faire de la surcharge d'opérateurs et réutiliser les symboles $+, -, *, /$, qui devront être désambigüisés lors de la phase de vérification de types. Cela nécessitera donc de modifier la signature des fonctions de vérification de types.

7.2 Instruction d'affichage (2 points)

Vous rajouterez aux expressions de `SimpleML` une opération `print_int` affichant un entier. Le type de retour de cette fonction sera `unit`, l'extension 7.0 est donc pré-requise.

7.3 Fonctions récursives (2 points)

Vous rajouterez aux déclarations de fonctions de `SimpleML` la possibilité de définir des fonctions récursives, de la forme

$$\texttt{let rec } f(x_1 : \sigma_1, \ldots, x_n : \sigma_n) : \tau = \texttt{M}$$

qui permet d'utiliser l'identifiant f dans `M`.

7.4 Variables mutables (2 points)

Vous rajouterez aux expressions de `SimpleML` la possibilité de déclarer des variables dont la valeur peut être modifiée. Pour cela, vous rajouterez à la syntaxe des expressions :

— `let mut` $(x : \sigma) =$ `M in N` pour déclarer une telle variable mutable

— $x :=$ `M` pour modifier une variable mutable en la valeur résultant de l'évaluation de `M`. Cette expression sera de type `unit`, l'extension 7.0 est donc pré-requise.

Vous pourrez également annoter vos environnements de type pour vous assurer que seul les variables mutables sont effectivement modifiées.

7.5 Tableaux (2 points)

Vous rajouterez un nouveau type `intarray` au type `typ` des types de `SimpleML` , de même que les constructions suivantes aux expressions de `SimpleML` :

— `new_array` n permettant de déclarer un tableau de taille n contenant uniquement des 0.

— `M`[`N`] pour accéder au tableau vers lequel `M` s'évalue, à la position représentée par l'entier vers lequel `N` s'évalue.

— `M`[`N`] := `P` pour modifier le tableau vers lequel `M` s'évalue, à la position représentée par l'entier vers lequel `N` s'évalue, avec comme nouvelle valeur celle vers laquelle `P` s'évalue. Cette expression sera de type `unit`, l'extension 7.0 est donc pré-requise.

On pourra ainsi évaluer le programme `let` $x =$ `new_array` 100 `in` $x[0] := 3; x[1] := x[0] + 1; x[0] + x[1]$

7.6 Boucles (2 points)

Vous rajouterez aux expressions de `SimpleML` une construction `while M do N` évaluant l'expression `N`, de type `unit`, tant que l'expression `M`, de type `bool`, s'évalue vers `true`.

Pour que cette extension soit utile, vous aurez besoin de réaliser également les extensions des variables mutables et/ou des tableaux.